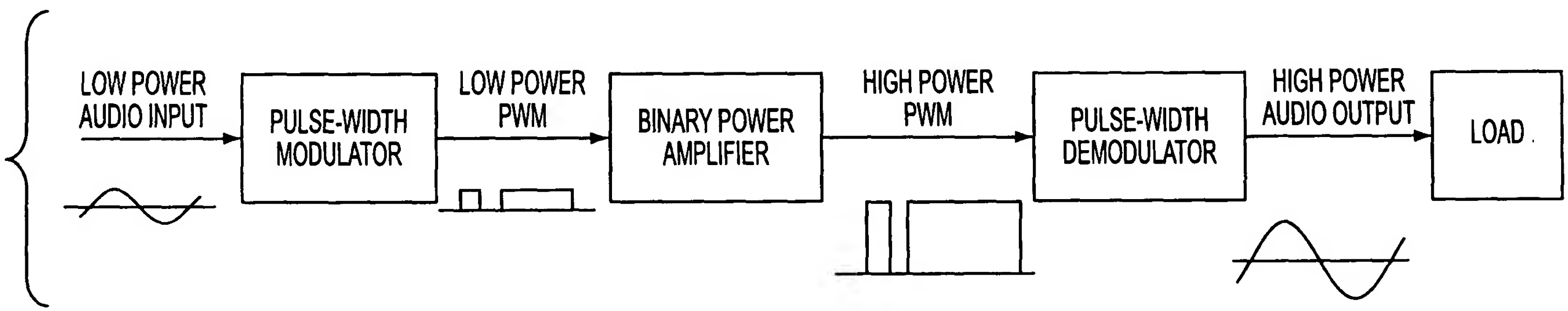

FIG. 1
PRIOR ART

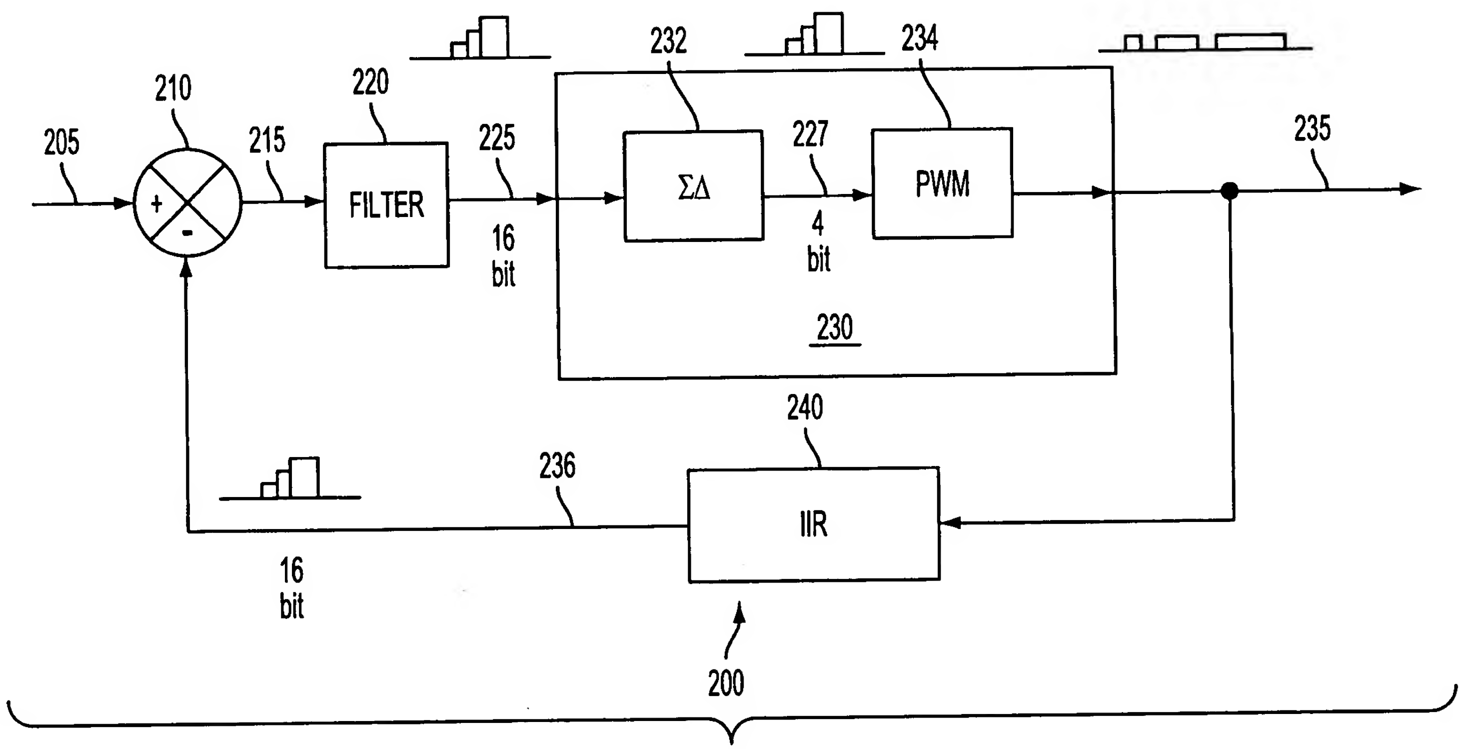
205
210
+
-
215
220
FILTER
225
16
bit
232
ΣΔ
227
4
bit
234
PWM
230
235
240
IIR
236
16
bit
200

FIG. 2

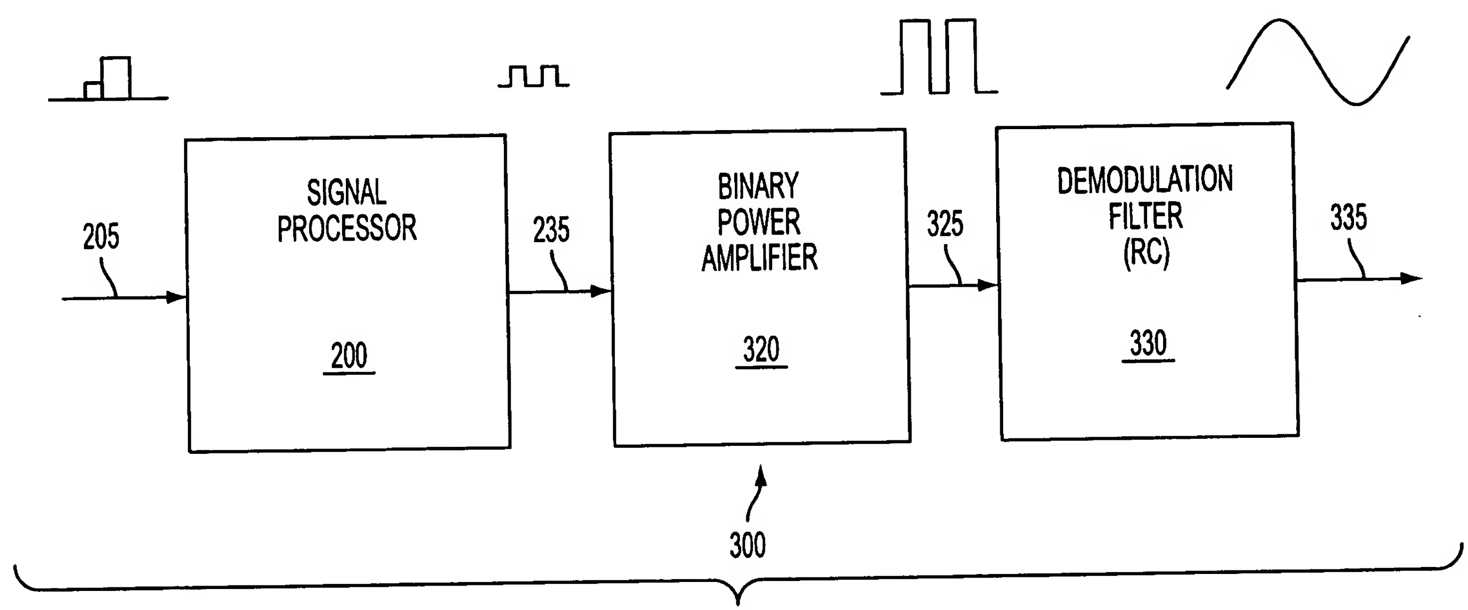
205
SIGNAL PROCESSOR
200
235
BINARY POWER AMPLIFIER
320
325
DEMODULATION FILTER (RC)
330
335
300

FIG. 3

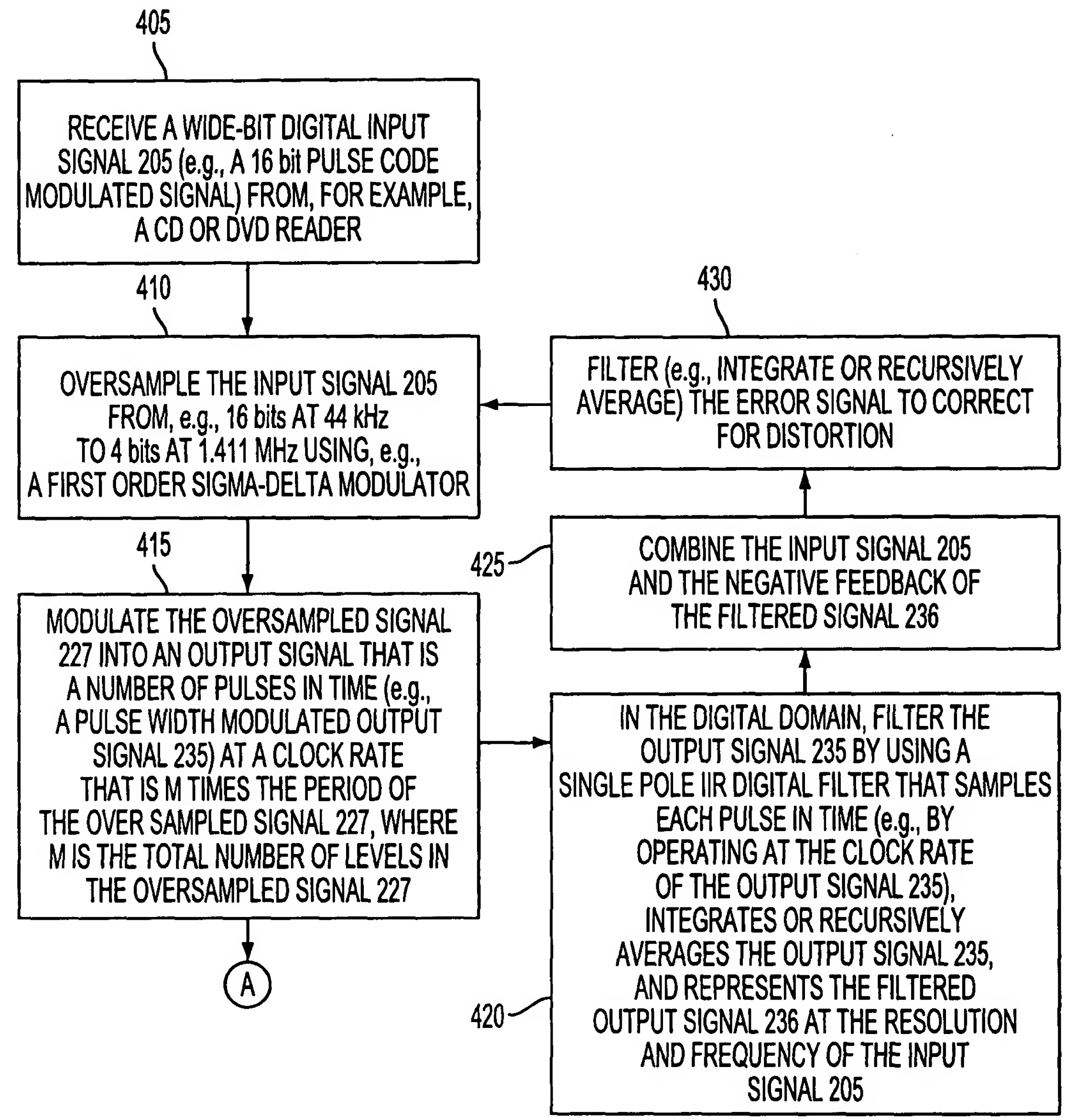
405
RECEIVE A WIDE-BIT DIGITAL INPUT SIGNAL 205 (e.g., A 16 bit PULSE CODE MODULATED SIGNAL) FROM, FOR EXAMPLE, A CD OR DVD READER
410
OVERSAMPLE THE INPUT SIGNAL 205 FROM, e.g., 16 bits AT 44 kHz TO 4 bits AT 1.411 MHz USING, e.g., A FIRST ORDER SIGMA-DELTA MODULATOR
415
MODULATE THE OVERSAMPLED SIGNAL 227 INTO AN OUTPUT SIGNAL THAT IS A NUMBER OF PULSES IN TIME (e.g., A PULSE WIDTH MODULATED OUTPUT SIGNAL 235) AT A CLOCK RATE THAT IS M TIMES THE PERIOD OF THE OVER SAMPLED SIGNAL 227, WHERE M IS THE TOTAL NUMBER OF LEVELS IN THE OVERSAMPLED SIGNAL 227
A
430
FILTER (e.g., INTEGRATE OR RECURSIVELY AVERAGE) THE ERROR SIGNAL TO CORRECT FOR DISTORTION
425
COMBINE THE INPUT SIGNAL 205 AND THE NEGATIVE FEEDBACK OF THE FILTERED SIGNAL 236
420
IN THE DIGITAL DOMAIN, FILTER THE OUTPUT SIGNAL 235 BY USING A SINGLE POLE IIR DIGITAL FILTER THAT SAMPLES EACH PULSE IN TIME (e.g., BY OPERATING AT THE CLOCK RATE OF THE OUTPUT SIGNAL 235), INTEGRATES OR RECURSIVELY AVERAGES THE OUTPUT SIGNAL 235, AND REPRESENTS THE FILTERED OUTPUT SIGNAL 236 AT THE RESOLUTION AND FREQUENCY OF THE INPUT SIGNAL 205

FIG. 4A

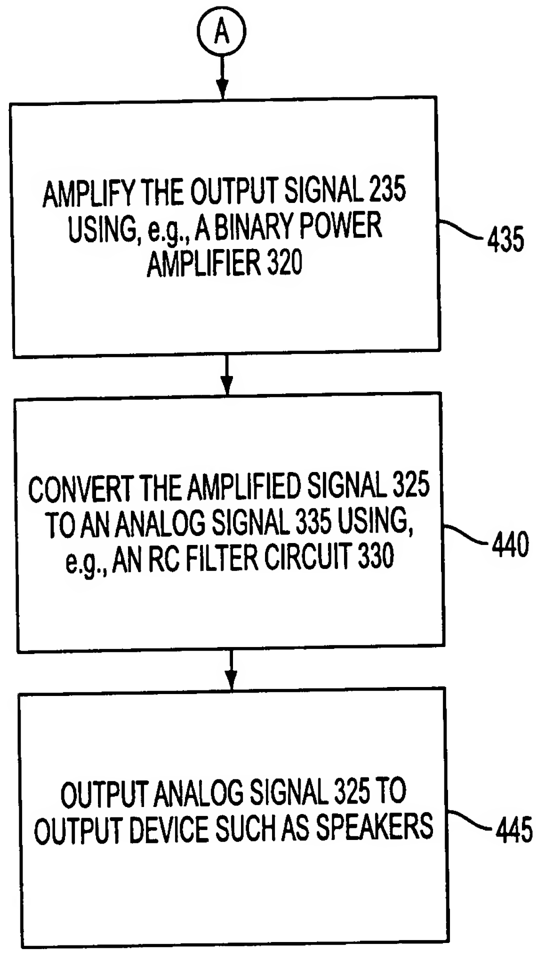
A
AMPLIFY THE OUTPUT SIGNAL 235 USING, e.g., A BINARY POWER AMPLIFIER 320
435
CONVERT THE AMPLIFIED SIGNAL 325 TO AN ANALOG SIGNAL 335 USING, e.g., AN RC FILTER CIRCUIT 330
440
OUTPUT ANALOG SIGNAL 325 TO OUTPUT DEVICE SUCH AS SPEAKERS
445

FIG. 4B

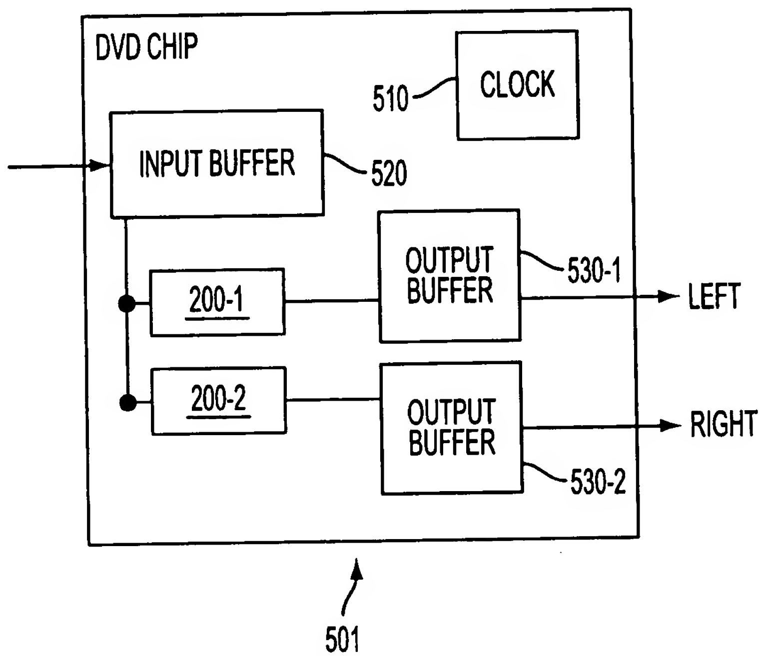
DVD CHIP
CLOCK
510
INPUT BUFFER
520
200-1
200-2
OUTPUT BUFFER
OUTPUT BUFFER
530-1
530-2
LEFT
RIGHT
501

FIG. 5A

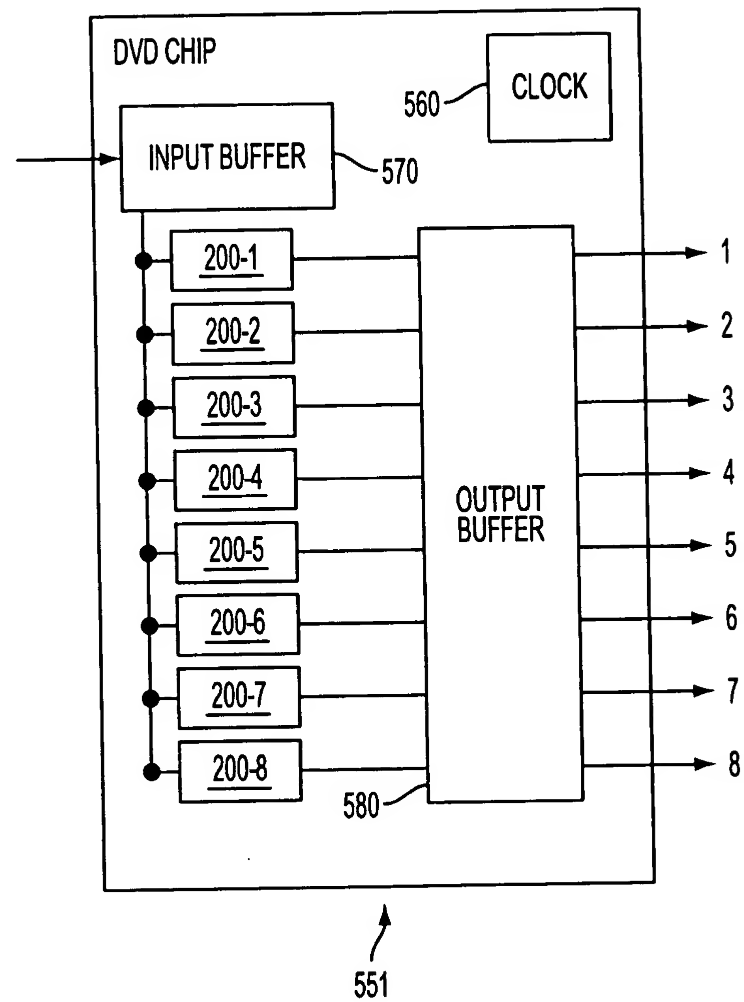
DVD CHIP
CLOCK
560
INPUT BUFFER
570
200-1
200-2
200-3
200-4
200-5
200-6
200-7
200-8
OUTPUT BUFFER
580
1
2
3
4
5
6
7
8
551

FIG. 5B

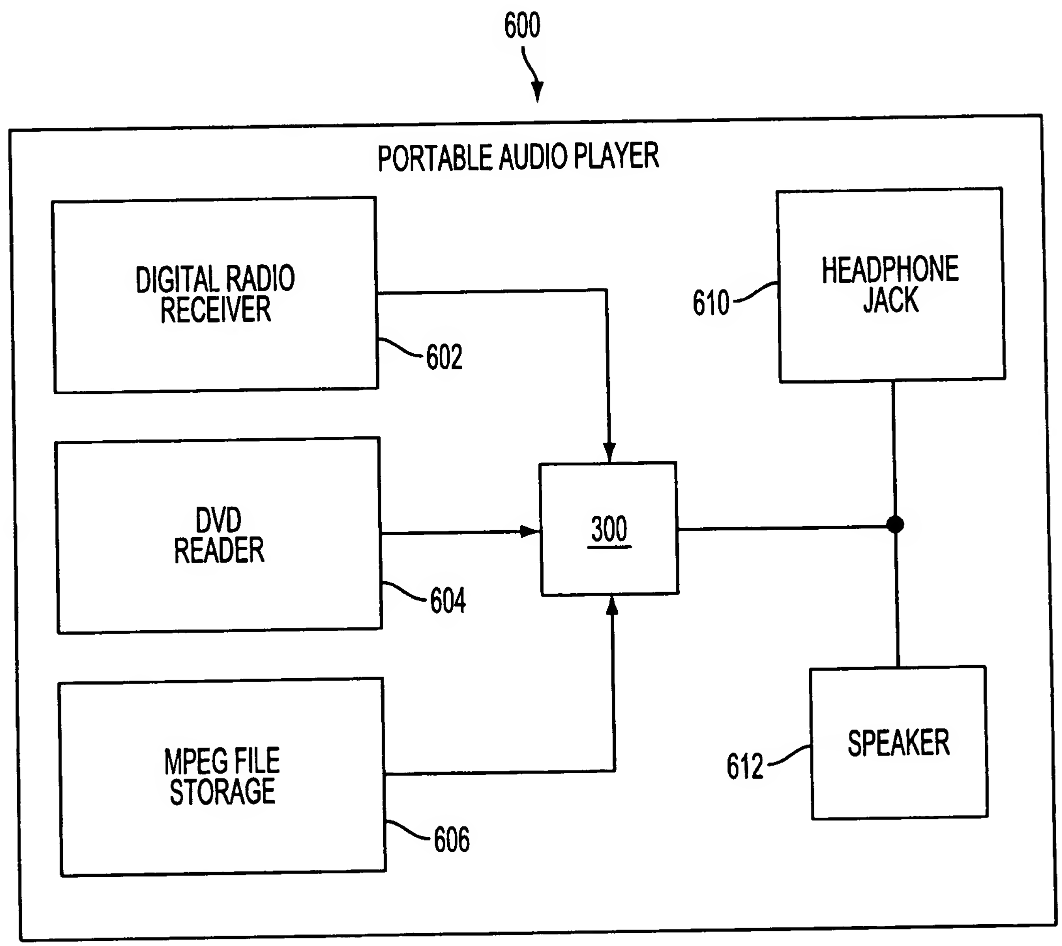
600
PORTABLE AUDIO PLAYER
DIGITAL RADIO RECEIVER
602
DVD READER
604
MPEG FILE STORAGE
606
300
HEADPHONE JACK
610
SPEAKER
612

FIG. 6

```
module Example_Embodiment   (clk,  clken,  reset,  phase,  in,  pwm) ;
    parameter WIDTH          = 24 ;            // Data path width
    parameter FACTOR         = 8 ;             // log2 of the divider in the ftr
    parameter PWMW           = 4 ;             // The number bits in the PWM
    parameter MI             = WIDTH - 1 ;     // Max index of bus
    parameter RST_TO         = (1 << MI) ;     // Reset state...
    input                    clk ;             // Main clock
    input                    clken ;           // A clock enable qualifier: Fclk is
                                               //   this enable expected to be about
                                               //   (16*32*Fs) = 24Mhz approx.
    input                    reset;            // Initialize asynchronous
    input  [PWMW : 0]        phase;            // PWM Phase control
    input  [MI : 0]          in;               // The input audio data
    wire   [MI : 0]          sin;              // The input scaled to -1.16db (7/8)
    output                   pwm;              // The output pwm bit
    reg                      pwm;
    wire                     pwma;             // Internal asynchronous pwmbit
    wire   [MI : 0]          fb;               // Feedback quantity..
    wire   [WIDTH : 0]       err;              // Error signal from differencer
    wire   [MI : 0]          serr;             // Scaled error value
    reg    [MI : 0]          int;              // Integrator in FB loop...
    wire   [WIDTH : 0]       sum;              // The actual sum prior to clip
    wire   [MI : 0]          nxt_int;          // Next state of clipped integrator
    wire   [MI : 0]          nxt_int1;         // Intermediate next state
    wire            force_all_one;             // Overflow and underflow bits
    wire            force_all_zero_b;
    // The forward path through the PWM to the output stream:
    pwm # (WIDTH, PWMW) pwmi (clk,  clken,  reset,  phase,  int,  pwma) ;
    // The feedback path through the filter:
    ftr # (WIDTH,  FACTOR)  ftri  (clk,  clken,  reset,  pwma,  fb) ;
    // This is one example way to clip an integrator - namely,  process one
    // extra bit and look for a difference in the extra bit and the
    // sign bit like this.
    assign #3   sum                    = { serr[MI] , serr }
                                       + {  int[MI] , int  };
    assign #0   force_all_one          =  sum[MI] & ~sum[MI+1] ;   //  Hi => OVF
    assign #0   force_all_zero_b       = (sum[MI] | ~sum[MI+1]) ;  //  Lo => UFL
    // This next line forces the nxt_dout1 bus to all 1 if force_all_one
    // is true,  forces it all 0 if force_all_zero_b is false...
    assign #0  nxt_int1  =  (  {~sum[MI] , sum[MI-1 : 0] }
                            | { WIDTH { force_all_one } } )
                            & { WIDTH { force_all_zero_b } } ;
    //  ..  and then just make the signed version..
    assign #0 nxt_int = { ~nxt_int1 [MI] , nxt_int1 [MI-1 : 0] } ;
    // Create the reduced amplitude input; in some cases, it may be advantageous
    // to drop the signal so that peaking at 15k is not distorted..
    assign #0 sin = in - { { 3 { in[MI] } } , in [MI : 3] } ;   //   ie in - 1/8 in
    // Now create the integrator in the loop .. first assign the
    // error gain: in this case, about 3/32
    assign #1    err   =  sin - fb;
    assign #1    serr  =  { { 4 { err[MI] } },err[MI : 4 ] }        // ie 1/16
                       + { { 5 { err[MI] } } ,err[MI:5] } ;         // + 1/32
    always @ (posedge clk or posedge reset)
        if (reset)
            begin
               int    <= RST_TO;
             pwm <= 1'b0;
            end
            else if (clken)
            begin
               int    <= nxt_int;
             pwm <= pwma;
            end
endmodule // Example_Embodiment
```

FIG. 7A

```
// ========================= PWM CELL ============================
// This is an example cell that accepts a parallel input bus and creates the
// output PWM stream...
module pwm (clk, clken, reset, phase, in, out) ;
   parameter WIDTH        = 16;              // Data path width
   parameter PWMW         = 4;               // The number bits in the PWM
   parameter MI           = WIDTH - 1;       // Max index of bus
   parameter PI           = PWMW - 1;        // Max index of phase bus
   parameter PMAX         = ( (1 << PWMW) - 1) ; // Max phase count
   input                  clk;               // Main clock
   input                  clken;             // A clock enable qualifier: Fclk is
                                             //  this enable
   input                  reset;             // Initialize asynchronous
   input  [MI:0]          in;                // The input state
   output                 out;               // The output state
   input  [PWMW:0]        phase;             // This is used to allow the clk to
                                             //   run faster than the BRM so that
                                             //   the multi levels are converted
                                             //   to pulse widths. The MSB
                                             //   controls the odd/even phase, the
                                             //   other bits the PWM
   wire [PI:0]              pwm;             // Pulse width required..
   wire                   pwmi;              // .. increment by one of above
   wire [PI:0]              sum;             // Temporary - used in PWM..
   wire                   pmax;              // Phase MSB is about to change
   wire                   pdoe;              // Enable to the pdo
   // Asynchronously find when the phase MSB is about to change:
   assign                 pmax = (phase[PI:0] == PMAX) ? 1 : 0;
   // Enable the pdo only when the phase MSB is about to change and
   // when the input is enabled - this generates the pdo clken signal:
   assign                 pdoe = pmax & clken;
   // Modulate the input bus into the PWM Width using something
   // very similar to the pdo cell again, but note that this generates
   // an output bus of width PWM width (and note this only runs at a
   // fraction of the input clock rate)..
   pdow # (PWMW, WIDTH) pdop (clk, pdoe, reset, in, pwm, pwmi) ;
   // Now generate the pulse output by comparing the phase
   // count to the multi-bit output, however this comparison depends
   // upon the phase MSB.. Lisp code for reference:
   // (if (<= (if MSB (1+ phase) (- dith phase)) pwm) 1 -1)
   // A simple equivalent can be found by looking at the carry
   // output of the following expression:
   assign {out,sum} = ( { { PWMW { phase [PWMW] } } } ^ phase[PI:0])
                    + pwm + pwmi;
   // Thus the output variable (out) is asynchronous - it occurs
   // shortly after the clock.
endmodule // pwm
```

FIG. 7B

```
// ========================== FILTER CELL ============================
// This is an example filter cell for the Example_Embodiment application,
// which includes a single bit in the feedback input. It's a
// simple IIR single pole filter like this: y <= y + a(x-y) where a is
// 1/ (2^FACTOR)
module ftr (clk, clken, reset, in, out);
    parameter WIDTH        = 16;          // Data path width
    parameter FACTOR       = 9;           // The log2 of the divider
    parameter MI           = WIDTH - 1;   // Max index of bus
    input                  clk;           // Main clock
    input                  clken;         // A clock enable qualifier: Fclk is
                                          //  this enable
    input                  reset;         // Initialize asynchronous
    input                  in;            // The input state
    output [MI:0]          out;           // The output state
    reg    [MI:0]          out;
    always @ (posedge clk or posedge reset)
        if (reset)
            begin
                out <= 0 ;
            end
        else if (clken)
            begin
                out <= out
                    - { { FACTOR { out [MI] } } , out [MI:FACTOR] }
                  + { { FACTOR+1 { ~in } } , { MI-FACTOR { in } } } ;
            end
endmodule // ftr
```

FIG. 7C

```
//========================== PDOW CELL =============================
//This cell is essentially a first order ΣΔ modulator - it creates
//an output word (out) and a bit (inc) indicating the word should be
//incremented by one to minimize the noise. There is one other
//circumstance to attend to here - the 'in' may be clocked at
//a rate that differs from this clock so that the following registers the input on
//the enabled clock of this cell..
module pdow (clk, clken, reset, in, out, inc);
   parameter M  = 4;                // The number of bits that are
                              // "dithered"
   parameter N   = 16 ;         // The width of the input number
   parameter NI  = N - 1 ;       // Max index needed in width N
   parameter MI  = M - 1 ;       // Max index needed in width M
   parameter ME = NI - M ;       // max index of the residue: this is
                              // the max index of the quantity
                              // that is accumulated in the
                              // modulo N error accumulator
   parameter RS = N - M;         // The amount to right shift the din
                              // bus
   input          clk;           // Main clock
   input          clken;         // A clock enable qualifier: Fclk is
                              // this enable
   input          reset;         // Initialize asynchronous
   input  [NI:0]  in;            // The input state
   reg    [NI:0]  rin;           // Registered input state
   output [MI:0]  out;           // The output state
   output         inc;         // The SD bit itself
   reg    [ME:0]  state;         // The local state of the modulo n error
   wire   [ME:0]  state_nxt;     // next state of the modulo n error
   //Just add the LSBs of the input to the running total in state,
   //allow state to overflow and keep track of the overflow bit in
   //the inc wire:
   assign  {inc, state_nxt} = {1'b0, state} + {1'b0, rin[ME:0] };
   //The output is just the msbs that are not being accumulated in
   //the state, plus 1 if the state has overflowed - as indicated in
   //the inc bit...
   assign   out = { ~rin[NI] , rin[NI-1:RS] };
   //clock like this:
   always @ (posedge clk or posedge reset)
     if (reset)
      begin
       state  <= 0;
       rin    <= 0;
      end
     else if  (clken)
      begin
       rin    <= in;
       state <= state_nxt;
      end
 endmodule // pdow
```

FIG. 7D

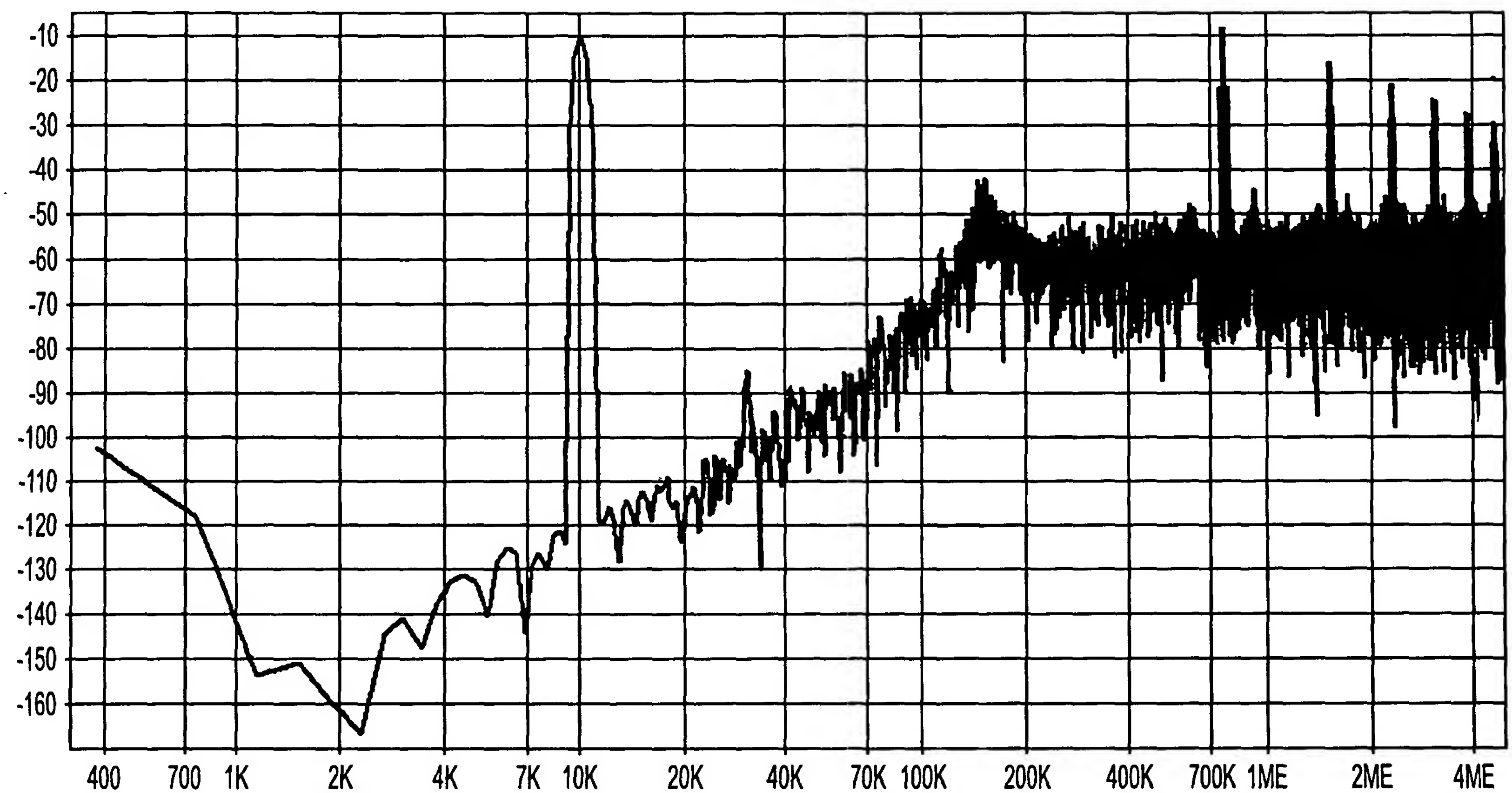
-10
-20
-30
-40
-50
-60
-70
-80
-90
-100
-110
-120
-130
-140
-150
-160
400
700
1K
2K
4K
7K
10K
20K
40K
70K
100K
200K
400K
700K
1ME
2ME
4ME

FIG. 8

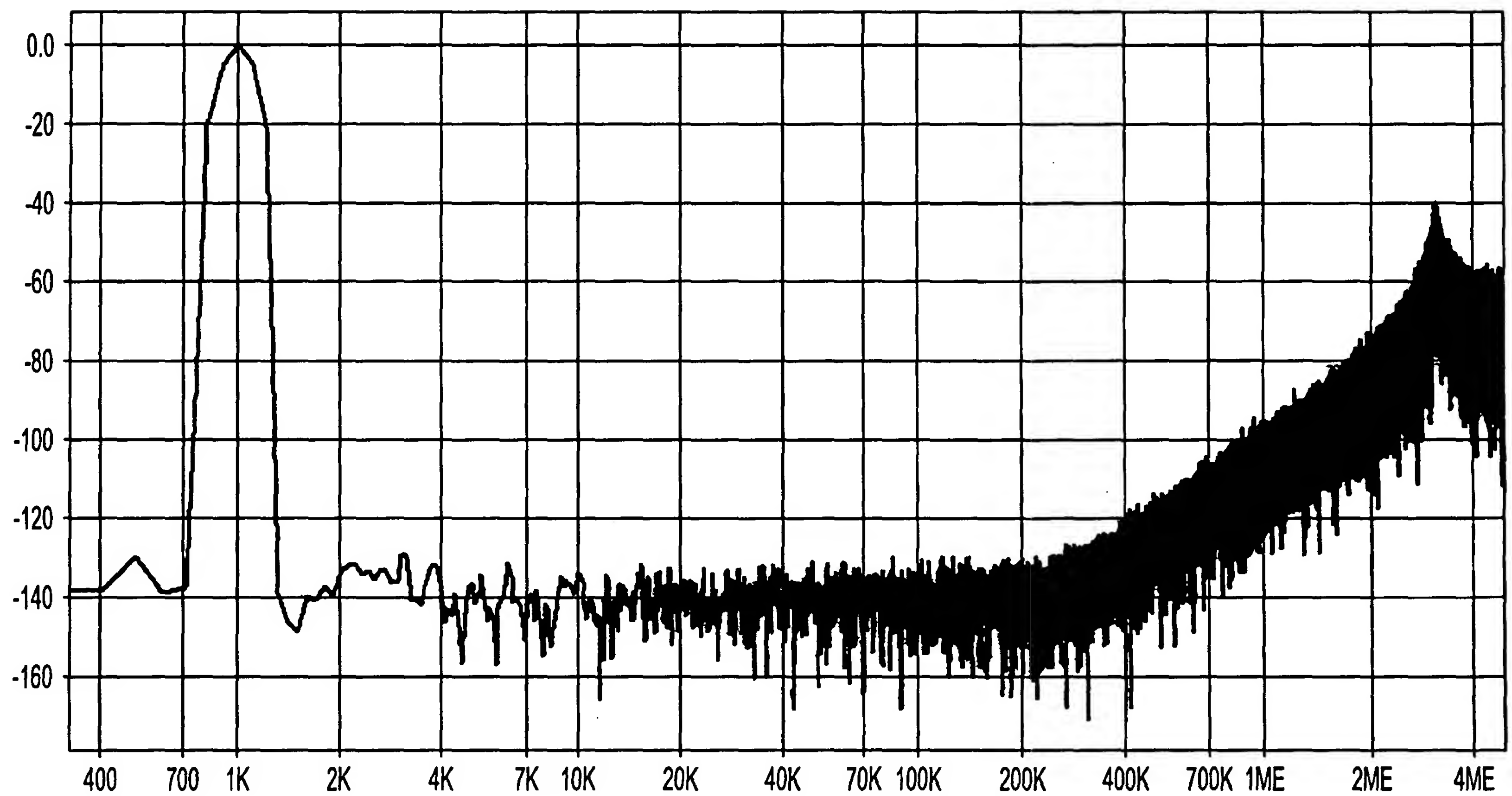
0.0
-20
-40
-60
-80
-100
-120
-140
-160
400
700
1K
2K
4K
7K
10K
20K
40K
70K
100K
200K
400K
700K
1ME
2ME
4ME

FIG. 9

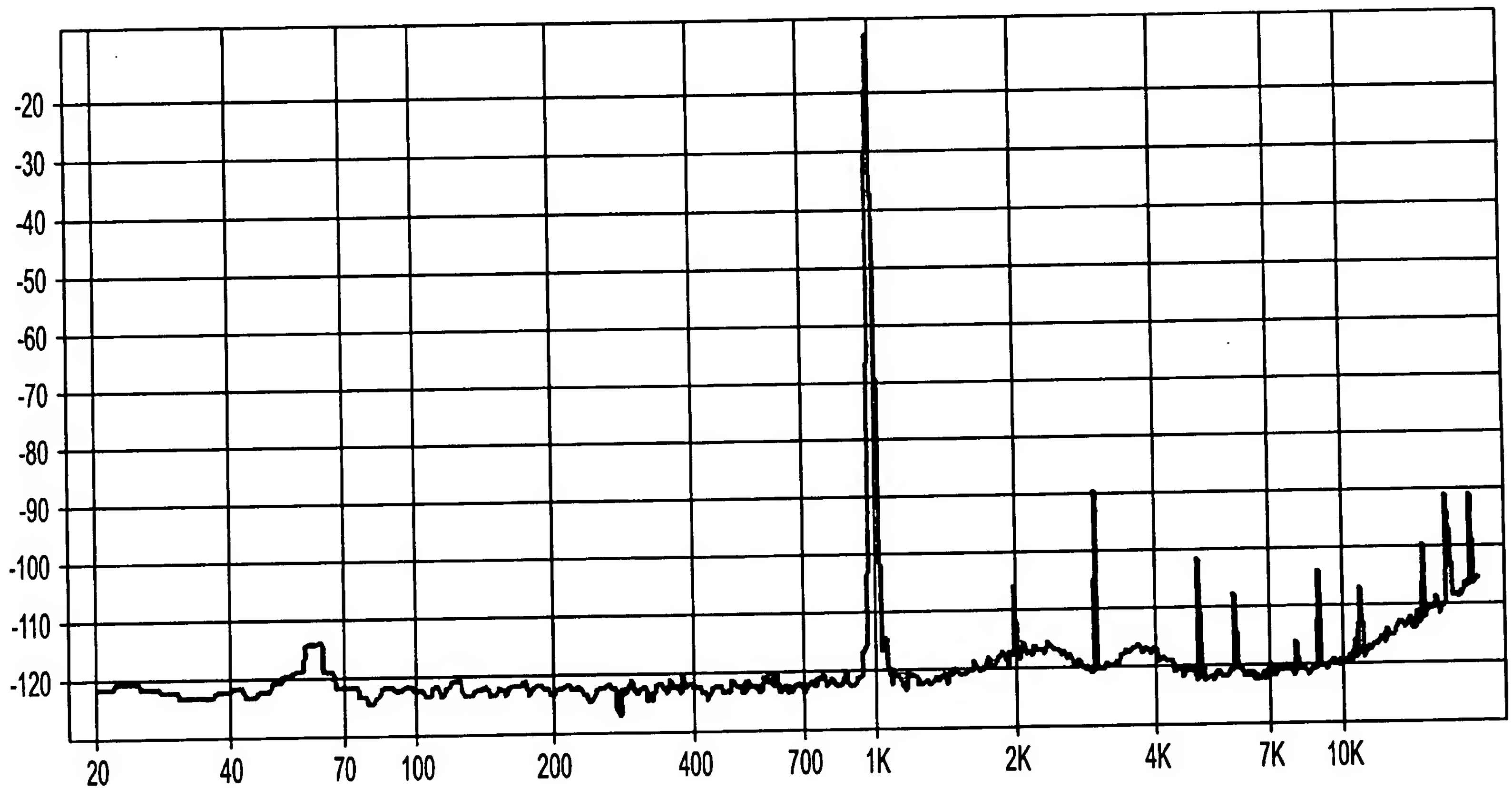
-20
-30
-40
-50
-60
-70
-80
-90
-100
-110
-120
20
40
70
100
200
400
700
1K
2K
4K
7K
10K

FIG. 10